Technical **Document**

rpslweb: Generating Insightful Data from Internet Routing Information

DCC009 Oriented Project in Computing II

Federal University of Minas Gerais ***2025H2***
Bernardo Reis de Almeida *(2021032234)*
Ítalo Fernando Scotá Cunha

1. Introduction

It is undeniable that the Internet has made fundamental contributions towards society. Currently gathering 5.67 billion users, almost 70 percent of the global population [5], this interconnected network of devices has enabled otherwise unimaginable applications in various fields, ranging from remote surgical operations in medicine, to data collection and forecast in logistics and real-time news dissemination in social media, among others.

In order to reach its current dimension, the Internet is organized as a hierarchy of networks [3], whose top-level is occupied by "autonomous systems" (AS): big collections of routers, links, and prefixes under the same administrative entity (e.g. a university, a government, a private corporation, or an "Internet Service Provider" (ISP)). These systems must exchange information regarding their reachable address prefixes and known routes for the purpose of allowing the existence of a global fully-connected network of devices. The exchange of such information, influenced by technical, political and economical interests, is organized through their routing policies, a set of rules that dictates how an AS will import and export data in relation to other ASes. These rules can, for example, determine how the information from one AS will flow towards another, perhaps to avoid routing traffic through some of them, and may be kept private by their authoring company.

Even though sharing routing policies is not an obligation, the information contained within them has immense value towards coordination and troubleshooting on the Internet [2]. Based on this, there exists an open initiative aimed at collecting, organizing and sharing this data in the form of the "Internet Routing Registry" (IRR), a publicly-available distributed database that stores information associated with routing policies [4]. These policies are described using the "Routing Policy Specification Language" (RPSL), an object-oriented specification language designed with the motivation of building a global view of the Internet's routing system, contributing to the maintenance of its integrity [1].

Network operators can use the information contained within the IRR, for example, to validate the origins of address prefixes announced by the "Border Gateway Protocol" (BGP), the most commonly used protocol to exchange routing data between ASes, creating filters that ensure the correctness of the information received, ultimately contributing towards Internet security [7]. Researchers can also use routing policies, for instance, in order to infer various details about the infrastructure and behaviour of the global network of computers, such as commercial AS relationships, sibling ASes and peering information, ultimately leading to a better understanding of it as a whole [2].

There are some problems, however, in directly using the data contained within the IRR. First, RPSL is a complex language, covering recursive structures and indirect references which make it difficult to directly interpret the meaning of some objects. It also makes

available an extensive syntax whose usage varies significantly and is prone to human error [2]. Second, being a distributed, collaborative-maintained database composed of several different independently-administered repositories, the IRR itself lacks a strong degree of reliability. Different repositories may employ different validation procedures with different degrees of security, while there seems to be a lack of coordination between them, as well as users may register information in many repositories only to maintain some up-to-date, resulting in possible inconsistencies across the database as a whole [7, 8].

Following these issues, but also the usefulness of such data, there have been some works in the literature that tackle these two aspects. In 2024, an article titled "RPSLyzer: Characterization and Verification of Policies in Internet Routing Registries", written by Sichang He, Ítalo Cunha and Ethan Katz-Bassett, proposed a parser that converts policies specified in RPSL into an intermediate and easier to deal with representation. The article also covers a comparison between the parsed information and BGP dumps, identifying strict matches, gaps, and syntax and semantic errors [2]. In terms of producing new insights, the article "IRR-Based AS Type of Relationship Inference", by Zulan et al., for example, establishes a series of heuristics that allow the inference of the type of relationship between two ASes based on their routing policies. It also defines metrics to evaluate the reliability of such heuristics and compares the results with ground truth, manually labeled datasets, revealing a reasonable degree of accuracy [13].

This situation, therefore, presents an opportunity for the development of a tool that extracts data from the IRR, processes it and generates useful and valuable insights that can help towards various of the tasks previously mentioned, mainly network operation and research. Such a tool could create operational representations of routing policies based on RPSLyzer and implement algorithms and methodologies proposed by research papers on top of them in order to generate new information. Ultimately, we believe that it could be of assistance towards the construction of a better Internet by assisting operators in troubleshooting, management and traffic engineering tasks.

$rpslweb$ is an initiative aimed at fulfilling this opportunity, making available useful information based on data contained within the IRR, empowering the parsing capabilities of RPSLyzer and generating new insightful data on top of it. This is achieved through a web application modeled after a search engine, with which a user can search various high-level information regarding routing policies, mainly associated with autonomous systems, autonomous systems sets, addresses prefixes and route sets, including relationships, announced prefixes, possible overlaps, as well as set membership.

This document aims at being a technical specification of $rpslweb$. First, some background and theoretical concepts necessary to comprehend the rest of the document will be presented in the section [Background and Foundations]. Next, the theory behind the main form of analysis built on top of IRR data will be presented in the section [Analysis: Type of Relationship Inference]. The features and design aspects of the platform will be presented in the section [Layout and Features], while the architecture of the system will be described in the following section, [Architecture]. The section [Deployment] will contain details pertaining to how the application is deployed and how it can be accessed. Relevant associated work found in the literature and similar tools will be presented and briefly discussed in the section [Related Work]. Finally, the section [Conclusion] will summarize the main results, and the section [Future Work] will discuss possible future developments for this project.

2. Background and Foundations

In the following subsections, many theoretical concepts and related foundations relevant to the current project will be addressed in more detail.

2.1. Background on Internet Routing

In order to reach the dimension it has reached, the Internet has had to face many problems mainly related to scalability. That is, given billions of individual devices, how to make a system capable of allowing them to interconnect onto one another in a fully connected manner and under many assumptions, which may be beyond technical, including political and financial interests. Not only that, but a connection which is sufficiently fast, reliable and that has the potential to accommodate constant and dynamic changes.

One of the main strategies adopted by the Internet was the use of a hierarchy [3]. On the bottom line, there are individual nodes, that is, single machines that connect to the network, including mobile devices, laptops, desktops and such. On top of that, those nodes are aggregated into smaller networks, usually based off of a single technology and using direct switching devices as means of connection. One level above, there are larger area networks, usually composed of sets of different technology based networks, which communicate and exchange messages through routers and associated protocols. Finally, on the top-level of the hierarchy, there are domains: autonomously directed corpus of internetworks controlled by a single administrative entity, which may be a provider, a government or a corporation. For these top-level networks is designated the term "autonomous system" (AS). As a consequence of this hierarchy, the problem of routing messages — as a proxy for the problem of establishing a global internet connection — can be digested between levels, turning it into a more feasible and addressable challenge.

2.2. Autonomous Systems and Interdomain Routing

In general, the problem of routing messages throughout the Internet can be divided into two main subproblems: intradomain routing — between networks under the same domain — and interdomain routing — between autonomous systems representing different domains [3].

For the former, a single administrative entity is in control of the whole infrastructure and, thus, many assumptions can be made in regard to the levels of trust and reliance on routing hardware. Precise metrics about traffic and performance can also be directly obtained. Protocols associated with intradomain routing, such as OSPF and RIP, usually employ some mechanism to distribute topology and performance information among routers, enabling the cooperative decision on shortest paths between nodes [3].

For the latter, on the other hand, information needs to transit between domains under different administrative entities, that is, trust is no longer a guarantee. In this case, routing does not aim directly at finding the shortest path within a well-known topology, but instead, it is directed towards finding any path that is loop-free and compliant with all the intermediary ASes' policies. A policy, in general terms, captures the interests of a given AS in the form of a specification of how received routes should be learned and how known routes should be announced. As an example, an AS X could define a policy such as "accept all routes received from AS Y that do not pass through AS Z" [3].

"Border Gateway Protocol" (BGP) is one of the main protocols developed for interdomain routing. As mentioned before, its goals are the discovery of loop-free compliant

routes between participant ASes. In order to achieve that, ASes that wish to partake in the protocol must employ border routers implementing it. These routers will contain the policies specified by their AS and speak with one another — hence, they are usually called "BGP speakers" —, exchanging route information in the form of complete enumerated paths. These paths associate network prefixes with a sequence of ASes that may be used to reach them. Once a path is discovered, the BGP speaker is able to verify for loops and compliance with the specified policies before learning it and redistributing it among intradomain routers [3].

2.3. Routing Policy Specification Language

The "Routing Policy Specification Language" (RPSL) is an object-oriented specification language first standardized in 1999 [1]. It defines various kinds of objects, each of which stores data points related to routing policies, such as import and export rules, administrative contact information or address prefix declaration. A full RPSL specification consists of a set of said objects that describe the specific data regarding one's domain.

Each object comes from a class, which defines the set of attributes that may be associated with them. In RPSL, there are various kinds of classes, including those related to contact information — "mnter", "person", "role" — and those focused on policies and associated data — "route", "aut-num", "inet-rtr" and set classes. A complete specification is organized in text form as a group of objects, each composed of attribute-value pairs, each of which is placed in one line, with the attribute name and its value separated by the character ":", while a blank line separates different objects. An example of an object retrieved from the database hosted by "Réseaux IP Européens Network Coordination Centre" (RIPE NCC) is presented. This object defines an instance of the "person" class, which contains contact information in regard to the referenced person.

```
person:          Mario Sauerbier
address:         Thueringer Netkom GmbH
address:         Schwanseestrasse 13
address:         99423 Weimar
address:         GERMANY
phone:           +49 3643 213031
nic-hdl:         MS50939-RIPE
mnt-by:          MNT-TNETKOM
created:         2020-09-15T12:21:25Z
last-modified:   2020-09-15T12:21:25Z
source:          RIPE
```

The "aut-num" class may be the most important definition in RPSL's syntax. It specifies the routing policies associated with an AS. In particular, every object in the language is uniquely identified through a key — their first attribute —, which, in the case of the "aut-num" class, is an AS number — an RPSL data type that refers to the unique identifier of an existing autonomous system. Besides it, this class also has three main multi-valued attributes, "import", "export" and "default", which specify, respectively, route importing, route exporting and default route behaviour policies. As an example, the syntax of an "aut-num" object's import policy is given below.

```
import: from <peering-1> [action <action-1>]
        …
        from <peering-1> [action <action-1>]
        accept <filter>
```

The semantics of this import attribute are as follows. It imports from <peering-N> — which may be an AS or a set of ASes — all the address prefixes — the routes' destination IP addresses, which may be grouped into prefixes — that match <filter>, while executing <action-N> when doing so — <action-N> can be, for example, setting some parameters for the import process. Similar definitions exist for the export and default attributes, completing the full specification of the routing policies for a given AS [1].

One thing to note is that RPSL supports indirect references and recursive definitions. For example, the <peering> parameter of the previous import rule can be a reference to another "aut-num" object, representing an AS, to an "as-set" object, which represents a set of ASes, or even to a "peering-set", another RPSL class that defines peers, which in its own definition can refer to and include other peering sets. The point is, as mentioned before, RPSL specifications can be arbitrarily complex, which really reduces the practicality of its direct usage and increases the chances of operating errors [2].

2.4. Internet Routing Registry

The "Internet Routing Registry" (IRR) is a conglomerate of distributed databases that host routing policy information specified in the "Routing Policy Specification Language" [4]. It started in early 1995 as an attempt to make a global view of Internet routing available in order to improve the Internet's integrity [1], and it is still currently the most used reference material for routing security purposes, such as the creation of filters [7].

Being a distributed database, it is composed of 18 different autonomous hosts, each of which may be aligned with differing interests, such as commercial, governmental or non-profit. Every individual database may be accessed independently in order to read or store routing data. This data may be used, for instance, by network operators in order to create router and filter configuration. Other entities, such as transit providers and "Internet Exchange Points" (IXPs, places dedicated to the exchange of data between different ASes and providers), also have their uses for all this information [7].

The biggest problem that arises with the IRR is the lack of coordination and the big autonomy each of its member databases detain. Each one may have their own validation and authentication measures in place, as well as may keep different levels of maintenance and up-to-dateness. In the mean time, a single AS may register the same or different prefixes with various different registries at the same time, contributing to the complexity and synchronicity needs of the system as a whole. As a consequence, all registries have varied levels of trust concerning their correctness, consistency and the recency of their data [7, 8]. For example, a prefix that has changed ASes may be updated in certain databases while not in others, which may lead to incorrectness of configurations and potentially to security vulnerabilities. This situation is only intensified considering the extent to which the IRR is being currently used and to the variety of applications this use comes from [7].

2.5. RPSLyzer

"RPSLyzer" is an open-source tool developed by Sichang He, Ítalo Cunha and Ethan Katz-Bassett with the aim of parsing RPSL-based specifications into an interpretable representation. In more details, it parses RPSL policies collected from the IRR into an "abstract syntax tree" that captures their meaning in a more digested intermediate representation, formatted into a dictionary. It is not the first mechanism aimed at dealing with the RPSL language, but it is the most general and applicable, capable of interpreting 99.99%

of policies available in public repositories, as well as uncommon but supported structures, recursive references and indirect syntax [2].

RPSLyzer has demonstrated promising results in digesting policies and analyzing features of Internet routing that are captured by them. In particular, besides the parser, the original project also dealt with the problem of comparing interpreted policies against BGP dumps containing AS-paths, with the objective of identifying inconsistencies between them. In doing so, it identified strict matches — policies that are correctly configured and that are consistent with the actual routing information being propagated through BGP —, as well as a lack of information — ASes that do not make their policies publicly available, for example —, syntax errors — errors in code writing, which are captured during parsing — and semantic errors — explainable by what have been classified as common misuses of RPSL [2].

3. Analysis: Type of Relationship Inference

Under the motivation of inferring new insights from the data contained in the IRR, in this section, the form of analysis implemented within this project will be presented.

rpslweb does a type of relationship inference between autonomous systems based on their routing policies. That is, given a set of ASes, infer the relationship, if any, established between each pair. The algorithm implemented was proposed in the article "IRR-Based AS Type of Relationship Inference", by Zulan et al. [13]. There is a plethora of work in the literature regarding extracting data from the IRR and, in particular, inferring the types of relationships between ASes, but this one was deemed especially interesting because first, it utilizes only RPSL policies and, therefore, it was straightforward to adapt it to use RPSLyzer's output, and second, the algorithm itself is pretty simple, but has revealed very reasonable results [13].

The algorithm is based on three heuristics, each one of which uses a different type of RPSL object to infer a set of possible relationships per object. Following the nomenclature adopted in the article, each object from which a set of relationships can be inferred is referred to as "entity". After inferring the "raw" relationships from each entity, based on them, a reliability score is calculated at entity level and those with less than a certain configurable threshold are discarded. Finally, the inferred relationships are united, first at an heuristic level, choosing the one with the highest reliability if they can be inferred from both ends, and next, between heuristics, which is based on a majority voting algorithm.

Of the three proposed heuristics, only the first two were implemented in this project, since the third of them was based on information that was contained only in the raw RPSL specification of each autonomous system, and it was deemed that returning to this state of the data, after having already parsed it with RPSLyzer, was counterproductive. The implemented heuristics are presented in more detail next.

3.1. Import/Export Heuristic

The "Import/Export (I/E) Heuristic" utilizes the "aut_num" RPSL object, which is conveniently parsed by RPSLyzer. The entity, in this case, is an instance of this object, which describes the routing policies of the corresponding AS. Specifically, the heuristic is based on the import and export rules of the given AS.

Typically, providers will export all of their known routes to their customers, while customers will only export their announced address prefixes and of their clients to their

providers. This is based on the "valley-free" assumption of route propagation, which, in general terms, determines that ASes won't want traffic transitioning between two of their providers while passing through them as this would incur costs with no returns.

Based on this, the idea is that, from the rules defined by AS X, if there is an import rule with respect to AS Y in which any route is imported, and an export rule with respect to the same AS in which a set of route objects is exported, it can be inferred that AS X is possibly a customer of AS Y. The symmetric analogous exists in the case that any route is exported and a set of route objects is imported. If both import and export rules exchange a set of route objects, then it can be inferred that these ASes are peers. This heuristic can be summarized in terms of a single table, which is shown in Table 1.

Table 1. Import/export heuristic [13]

Role	Import	Export
Customer	ANY	Route Objects
Provider	Route Objects	ANY
Peer	Route Objects	Route Objects

Given the "aut_num" object of AS X, a set of relationships can be inferred by this process of pairing import and export rules and verifying their compliance with the heuristic. This can be repeated for all other entities, such that each one defines a set of relationships.

3.2. Set Heuristic

The RPSL specification allows the declaration of hierarchical AS set names, which is accomplished with the use of the symbol ":". For example, the name "AS123:AS-CUSTOMERS" first declares a namespace, in this case, referring to AS 123, and then the name of an AS set, within this space. This allows, for example, multiple ASes utilizing the same set names, which are then distinguished by the namespaces.

As exemplified above, some ASes utilize AS set object specifications to declare their customers, providers or peers. In practice, this usually manifests through hierarchical AS set names whose first component is the AS that declares them, and whose second component is the name of the AS set, which may include keywords that reveal its purpose.

Table 2. Common keywords for the set heuristic [13]

Relationship	Keywords
Provider/Customer (P2C)	downstream, downlink, customer, client, custs
Customer/Provider (C2P)	provider, upstream, uplink, backbone
Peer to Peer (P2P)	peer

Based on this, the idea of the set heuristic is to work with AS set objects as its entity and, for each one, verify if its name follows the presented pattern. If it does, then all the members of the set are inferred to establish the corresponding relationship — given by common keywords associated with each one — with respect to the AS that hosts the object. This list of members constitutes the inferred relationships from the entity represented by the

AS set. In Table 2, some common keywords associated with each type of relationship and used in the name of AS sets are presented.

3.3. Reliability Score

For each heuristic, once its whole set of relationships is determined, a reliability score is assigned per entity, which is calculated over the amount of relationships that agree between them. In particular, for each entity, each one of its relationships is labeled as "bidirectional" if it could also be inferred from the entity corresponding the other end of the relationship, and "agreeable" if, besides being bidirectional, the relationships match — that is, if "Customer" was inferred from one end, "Provider" should be inferred from the other, and so on.

Once all the relationships have been categorized, the reliability of an entity is defined as its proportion of agreeable and bidirectional relationships. For example, if there could be inferred 10 bidirectional relationships from a certain entity and, of those, 5 are agreeable, then its reliability score is 0.5. It is noted that, if the entity has too few relationships (less than a configurable threshold), it is deemed not reliable and its reliability is set to 0. After assigning a score to each entity, they can be filtered based on a configurable threshold. That is, entities that have a low reliability score can be excluded from the analysis.

3.4. Unifying the Results

In order to unify the results of all entities within each heuristic, each of its relationships is traversed and, if it could be inferred from two entities, the one with the highest reliability score is chosen. Otherwise, the only existent inference is chosen. The relationship inherits the reliability score of the entity from which it was inferred.

For the unification of relationships across heuristics, a simple algorithm based mostly on majority voting is used. The original algorithm had to be adapted, considering that, from the original three, only two heuristics were used. It was also simplified, in the sense that one of the rules was dropped since its implementation proved to be a bit complicated. Each relationship is traversed and evaluated according to the following rules:

1. **Majority vote:** if both heuristics classify the relationship in the same way, the reliability score is set to 1 and the relationship is added to the final set;
2. **Highest reliability:** if the heuristics do not agree, the relationship with the highest reliability score is chosen and added to the final set. In this case, the final reliability score is the one of the relationship itself;
3. **Default:** if the relationship has the same reliability score on both heuristics, the relationship from the I/E heuristic is chosen and added to the final set, as it is deemed more reliable;

The dropped rule established that, if the heuristics do not agree and they have the same reliability score, the relationship should be extracted from the entity with the most bidirectional relationships. In the way the algorithm was implemented in this project, the information regarding the amount of bidirectional relationships per entity was lost during their unification within each heuristic, such that it became a bit complicated to propagate the information in order to implement the rule. The final results seemed reasonable nonetheless.

After the unification of all heuristics, the final result is a JSON object, whose keys are AS numbers, and whose values are also JSON objects. The keys of the inner objects are the AS numbers of the ASes with which the outer ones establish a relationship with. The values of the inner objects are also JSON objects that contain the information regarding the relationship, including its type ("Customer", "Provider" or "Peer", with respect to the outer

AS), the bidirectionality status, the agreement status, among others, discussed further below. In Figure 1, a simplified example of the final JSON object is presented.

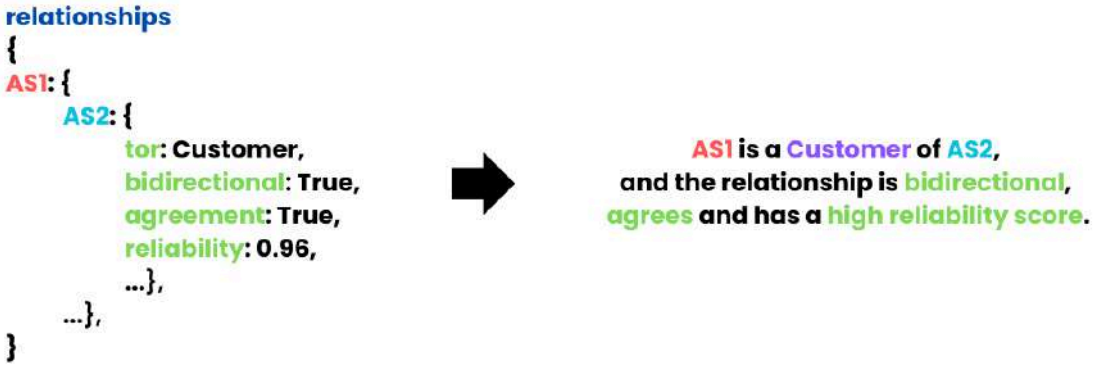

Figure 1. Simplified example of the result of the type of relationship inference

3.5. Extensions

Besides all the adaptations necessary to adjust the algorithm to the data produced by RPSLyzer, and also all the simplifications made, two main extensions were added to it.

First, as mentioned in the I/E Heuristic subsection, usually, customers and peers will export a predefined set of route objects consisting of their own announced address prefixes and those of their clients. In RPSL, various objects can be used in the place of a set of route objects, such as an AS number — meaning all the route objects originated by this AS — or an AS set — meaning all the route objects originated by its members. Originally, it seems that the article considers only AS numbers, a comma-separated list of AS numbers and an AS-set [13] in order to identify the relationship, but the algorithm was extended to also consider route sets in the way it was implemented in *rpslweb*.

Second, besides bidirectionality, agreement and the reliability score, two other metrics regarding the quality of the inferred relationship were added to the results. The first of them describes if the relationship is "representative". As mentioned previously, a client/peer will export a certain object to its provider/peer. If this object is the one this client/peer exports the most, the relationship is deemed representative, as it is common for ASes to utilize a certain object to refer to the routes they want to export to providers/peers, information which contributes to the reliability of the inferred relationship. The second of them describes if the relationship is "Internal" or "External", meaning if it was inferred from the given AS (that is, from its "aut_num" object or from one of the AS sets in its namespace), thus internal, or from its partner's end, thus external. Here, this information is used in order to filter only the external relationships of a given AS. It was deemed that, if a user is searching for a specific AS, it would be interesting to show all the relationships that could be inferred from the data associated with it, regardless of reliability score, but to show only relationships inferred based on information from other ASes if they passed the reliability threshold and, thus, were reasonably reliable. In other words, the filtering was done in a more fine-grained way.

4. Layout and Features

As mentioned, *rpslweb* is a web platform modeled after a search engine. On the main page, as shown in Figure 2, the user has access to a brief description of the project and also to a search box, on which it can input a query. This query can be one of four keywords: the

identifier of an autonomous system, that is, its AS number; the identifier of an autonomous systems set, that is, its name; the identifier of a route set, that is, also its name; or an address prefix of the form 255.255.255.255[/16]. For each type of query, relevant information pertaining to the corresponding type of object will be returned.

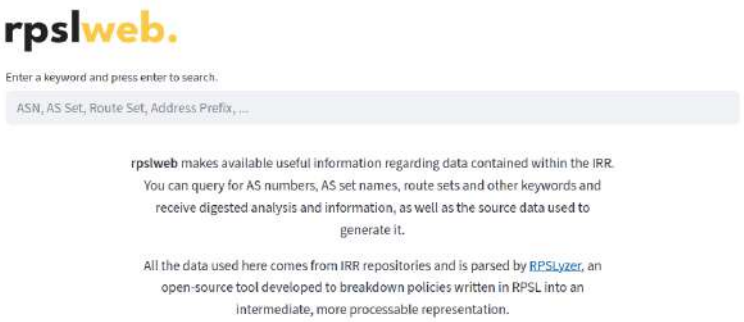

Figure 2. rpslweb's main page

4.1. Search for an Autonomous System

A search for an autonomous system will return five main blocks of information. Each block is presented in the form of a table with the corresponding relevant information. The table displays a limited number of entries and there is a navigation menu through which the user can navigate through the full list of results. A search bar is also made available, through which the user can filter the results to include desired keywords.

The first block of information is dedicated to some basic attributes regarding the autonomous system's corresponding object in RPSL. It includes its description, in which IRR database it was registered, contact information for technical and administrative affairs, among others. It also presents any remarks the maintainer of the AS has left within its specification, maybe including some more specific information. This can be seen in Figure 3.

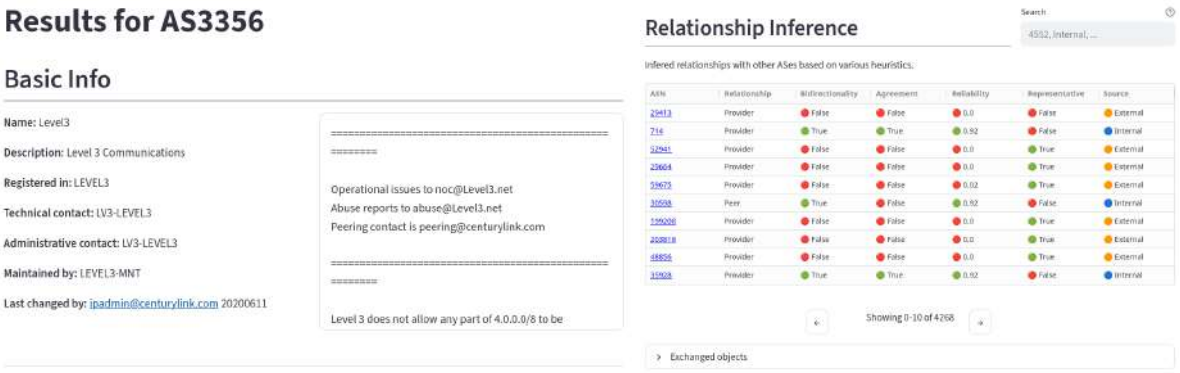

Figure 3. Basic information (left) and relationships (right) of an AS

Next, the lists of import and export rules defined by the given AS are presented. Both of these lists characterize its routing policies. Each one of the rules is defined through five attributes: the IP version for which it is applied (ipv4, ipv6 or any), its cast (unicast, multicast or any), the other AS to which the rule refers to (its AS number), the rule's filter (the set of exchanged addresses prefixes) and actions defined for the rule (a dictionary mapping RPSL actions). This is shown in Figure 4.

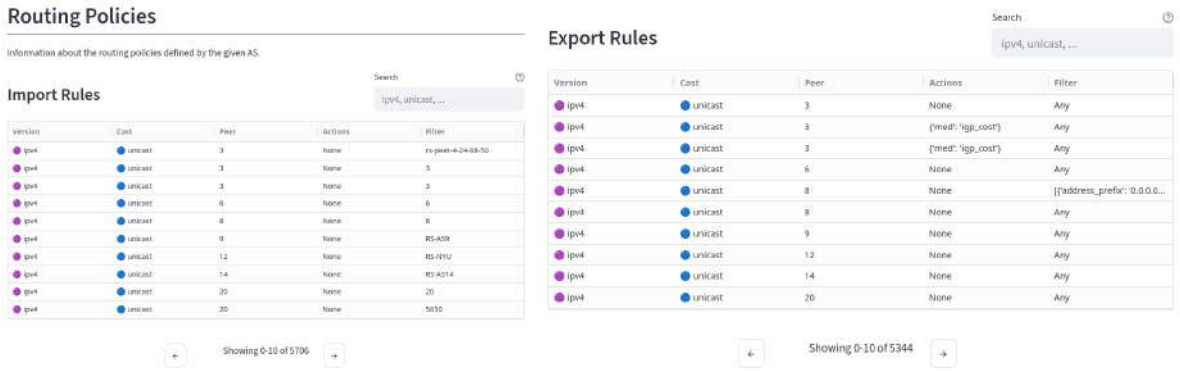

Figure 4. Import rules (left) and export rules (right) of an AS

Following this listing of the autonomous system's import/export rules, next, a list of possible relationships established with other ASes is displayed, as also shown in Figure 3. The way these relationships were inferred was discussed in more detail in the section [Analysis: Type of Relationship Inference]. As mentioned there, these relationships can be of three different kinds: customer, provider or peer, all with respect to the given AS. Besides the type of relationship and the AS with which it is established, five other pieces of information are also displayed. First, if the inferred relationship is bidirectional, that is, if it can be inferred from both ends. Second, if it agrees, that is, if it is bidirectional and if both inferred types of relationship match (for example, customer was inferred from one end, while provider was inferred from the other). Next, a reliability score for the inference is presented, revealing how much it can be trusted. Another reliability metric, the representativeness of the relationship, is also displayed. Finally, the end from which the rule was inferred (from the given AS or from its peer), its "source", is also provided.

For this particular block of information, a dropdown menu is also provided, which contains the most exchanged objects defined in the autonomous system's import and export rules. This information is relevant for the definition of the representativeness of the relationship, which is described as if the object exchanged between the ASes, from the customer to the provider (if applicable), is the most exchanged object of the customer.

Finally, two more blocks of information are presented, one regarding AS set membership, and other, registered route objects. For the former, a list of the sets of which the given AS is a member of is displayed, along with some of their basic information, such as their number of members. For the latter, besides the address prefixes registered, a list of other ASes that have also registered the same address prefix is provided, indicating if there are any possible overlaps in the database. These sections are shown in Figure 5.

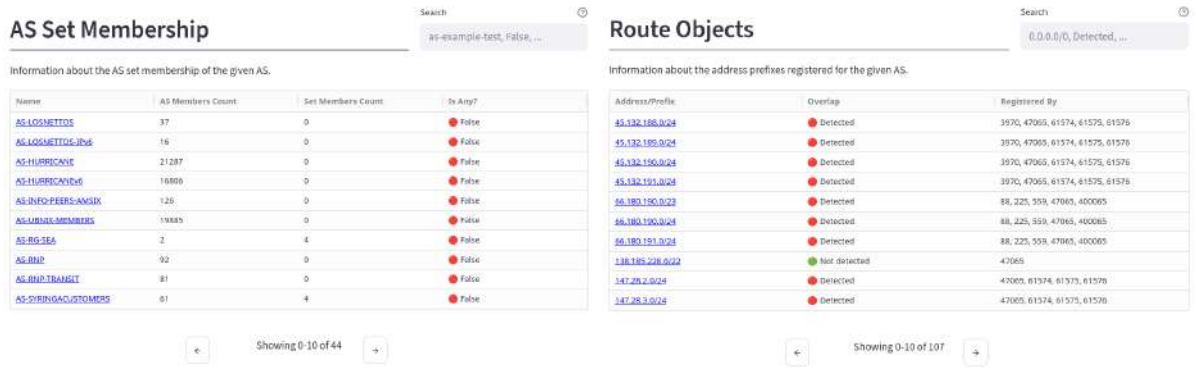

Figure 5. Set membership (left) and registered route objects (right) of an AS

4.2. Search for an Autonomous System Set

A search for an autonomous system set will return two blocks of information, as shown in Figure 6. First, analogous to a search for an AS, a block with some basic information referring to the RPSL specification of the set will be displayed, including its description and personnel contact information. Second, all of its members will be displayed in the form of a searchable table, including ASes and other AS sets.

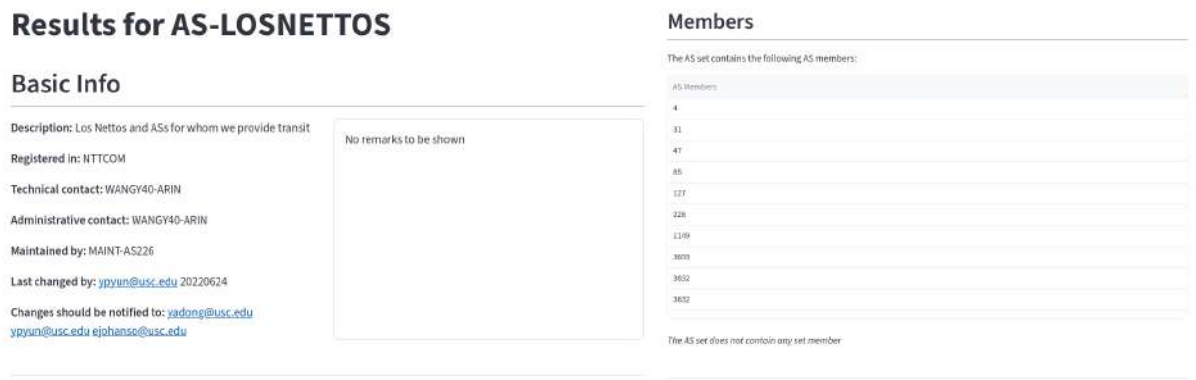

Figure 6. Basic information (left) and members (right) of an AS set

4.3. Search for a Route Set

A search for a route set will return analogous information as for a search for an AS set. First, a brief block with some basic information registered in the route set's RPSL object, such as its description and contact info. Next, a list with all the members of the set, with the difference that, here, each member is an address prefix or another route set.

4.4. Search for an Address Prefix

A search for an address prefix will return a single block of information: the list of autonomous systems that announce such address prefix, as shown in Figure 7.

Figure 7. Results for a search for a given address prefix

5. Architecture

rpslweb was developed entirely using the Python programming language. The whole system is composed of three main modules: a data processor, responsible for ingesting the data gathered from RPSLyzer, processing it further and storing it; a back-end, responsible for making this data available through a REST API; and a front-end, through which the user will be able to communicate with the system, making queries and receiving the results. A diagram illustrating a typical usage flow and the interactions between the modules is shown below.

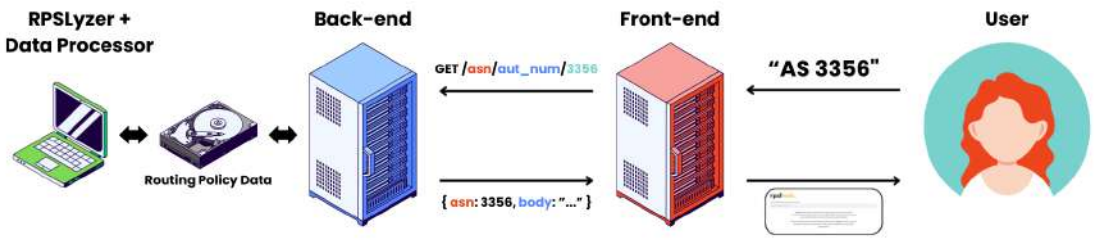

Figure 8. Flow of the system during typical usage

The data processor is a separate component in the sense that it acts independently of the back-end and the front-end modules. Its task is to further digest the data generated by RPSLyzer and organize it in disk in the form of a data repository for the rest of the system to use during its operation. The front-end is also unique in the sense that it follows an atypical architecture, given the framework with which it was implemented, which will be detailed further in the next sections. Following Figure 8, a user first accesses the web platform through its "Uniform Resource Locator" (URL), upon which the front-end server returns the main page. Next, the user inputs a query on the search bar and sends the request to the system, which proceeds to check the validity and the type of entity the query refers to, be it an AS, an AS set, a route set or an address prefix. Once identified, it reads the corresponding data from the disk and sends it back to the user in the form of a web visualization. In the following subsections, the inner workings of each module are further detailed.

5.1. Data Processor

The data processor is implemented as a Python module which can be run through the command line. It has two main purposes: further refining the data generated by RPSLyzer — which will be called the "processing" task — and executing the form of analysis implemented — which will be called the "analysis" task. On the rest of this subsection, the processing task will be further detailed, as the analysis task was already previously presented in the section [Analysis: Type of Relationship Inference].

Before the execution of the data processor itself, it is necessary to execute RPSLyzer in order to gather the raw data from the IRR and parse it into an intermediate representation. This process was not included directly into the data processor, but a Docker container was built to automate it and put the results in a shared volume. Further details of the full system deployment architecture are discussed in the section [Deployment].

Focusing on the processing task, it is divided further into four main subtasks, the first one of which being the "organization" subtask. Following the instructions for RPSLyzer's execution, present in its GitHub repository [6], it processes the raw data from the IRR in separated shards, resulting in various files following the same format, but corresponding to different pieces of the whole dataset. During the organization subtask, the data processor retrieves each shard and merges them into a single file, a JSON object containing all the intermediate representations of routing policies generated by the tool.

The output of RPSLyzer, the JSON object mentioned, contains six keys: "aut_num", "as_routes", "as_sets", "route_sets", "peering_sets" and "filter_sets". The value of each one of them is another JSON object containing information about their corresponding RPSL object, indexed by their identifier. For instance, take the object of the "aut_num" key. Its keys are AS numbers, and their values are new objects containing the information of the

corresponding AS, defined in its RPSL specification, but interpreted by RPSLyzer, such as its description, import and export rules. The other keys are defined in an analogous way.

The next subtask is the "restructuring" subtask, and, as its name suggests, it reestructures some of the data returned by RPSLyzer. In order to illustrate that, one of the most important pieces of information returned by this tool are the import and export rules for a given AS, which describe its routing policy. However, this information is organized in layers instead of in key-value pairs, which was deemed as more difficult to deal with for the purposes of *rpslweb*. In Figure 9, a diagram showing how this information is provided by RPSLyzer is presented.

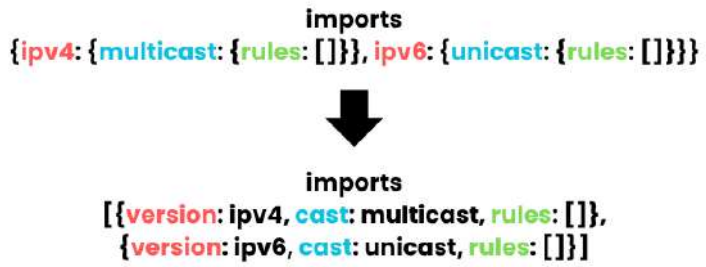

Figure 9. Example of the modifications applied by the "restructuring" subtask

As can be seen in the top half of Figure 9, an import or export rule is characterized by the IP version for which it was defined, its cast, and the actual list of rules. However, as given by RPSLyzer, each piece of information is organized in an "inner layer" of the JSON object, resembling a tree-like structure. What the restructuring subtask does in this case is convert this representation into a key-value model, effectively "flattening" the original JSON object, as can be seen in the bottom half of the image. Besides restructuring the import and export rules of each AS, this subtask also applies some form of flattening to other pieces of information, and also incur some minor modifications to other objects returned by RPSLyzer.

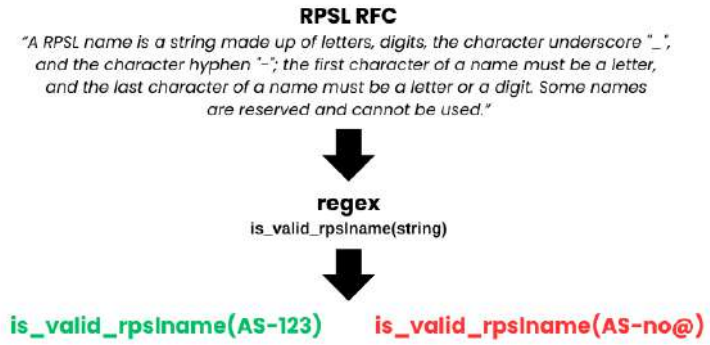

Figure 10. Example of a filter being applied by the "cleaning" subtask

The next subtask is the "cleaning" subtask. Again, as its name suggests, it filters out any inconsistencies found in RPSLyzer's output. As a matter of fact, one of the objectives of this tool is to identify inconsistencies in RPSL specifications, which are caught during its execution, registered and treated in some form, with the objective of generating consistent intermediate representations. However, during the development of this project, some errors which were not being caught were identified. As an example, in the definition of a route set, its members are separated into two main groups: address prefixes and name objects (such as other route sets). Some name objects were identified as having names corresponding to a list

of address prefixes, separated by commas, a misuse of RPSL that was not being caught by RPSLyzer and led to some problems during the data processor development.

Following the presented example, in general, the cleaning subtask passes over the data generated by RPSLyzer and checks the values of some keys, using regular expressions in order to verify if they comply with the norms established in RPSL's specification. If they do not comply, they are simply eliminated from the data. This is illustrated in Figure 10.

Finally, the last subtask is the "parsing" subtask. Its main purpose is to read through the whole output of RPSLyzer, now united, restructured and cleaned, and process this information, as to separate it into smaller, cohesive units. Taking the "aut_num" key of the tool's output as an example, it contains a JSON object whose keys are AS numbers and whose values are more JSON objects describing each AS. Inside these latter ones, there is information regarding their import and export rules, among others. This subtask is responsible for extracting this information into its own standalone JSON object, indexed by the identifier of the object it refers to — in this case, an AS number. The purpose of this is making important data deeply nested inside the original structure more readily accessible. An illustration of this process is given in Figure 11.

Figure 11. Example of the modifications applied by the "parsing" subtask

Once all the final JSON objects have been generated, the data processor module stores them in disk using Pickle, a Python module for binary serialization and deserialization. Each object is stored within its own binary file, which is a faster and more compact approach when compared to dealing with JSON files following a text format. This implementation was judged reasonable enough for the current purposes of the system.

The storage is handled by a separate class, which offers two main options of storage. First, simply write the received object in disk, following the Pickle format. However, most, if not all of the objects, are dictionaries indexed by identifiers, such that, sometimes, only the information associated with a single identifier is desired. If the object is too big, in order to avoid reading all of it into memory in order to retrieve a single key, it can also be stored as a folder — a "bucket" of sorts —, with files inside it named after each identifier and containing only its information. The storage class also implements a simple cache in memory, with a hard-coded limit and a random selection scheme for deleting old entries.

5.2. Back-end

The back-end is implemented as a REST API server using the framework FastAPI, in Python. Its inner workings are pretty straightforward, with the server being just an access point for each object created and stored in disk by the data processor module. Following the REST architecture, each object is associated with an endpoint, through which a GET operation can be made to retrieve the correspondent information.

As already mentioned, most objects produced by the data processor module are indexed by the identifiers of a certain entity, such that the information of only one of them is

usually what is being requested. In this way, each endpoint receives this key and returns to the user only the data referring to the desired entity, and not the whole object. Some endpoints whose contents are list-like — be it an actual list or a dictionary whose keys are identifiers — also offer the possibility of "paging", through which the user can specify a certain range off of which the corresponding content will be sliced and then returned. This is a matter of optimization and is used by the front-end to prevent too much data from being loaded at once, with options for the user to navigate between pages.

A final feature available with some endpoints is a search functionality, through which users can specify keywords and only the elements of the result containing them will be returned. The input to this search must be a comma-separated string of keywords, and the verification is done simply by converting the elements to strings and checking substrings, with some robustness in the form of white-space elimination and case insensitivity.

An example of a typical usage workflow is shown in Figure 12. It illustrates the endpoint for the "relationships" object, which contains inferred relationships. The keys of the object are AS numbers and their values are the corresponding list of relationships. Each relationship in the list is itself an object, with attributes representing the target AS (with which it establishes the relationship), the type of the relationship, its symmetry, among other information, as detailed in previous sections. Accordingly, the endpoint receives a key as a parameter, corresponding to the identifier of the AS of which the information is desired, as well as a range with which it will slice the list. It returns to the user the sliced list inside a JSON object, which also contains other meta-information, such as its size and limits.

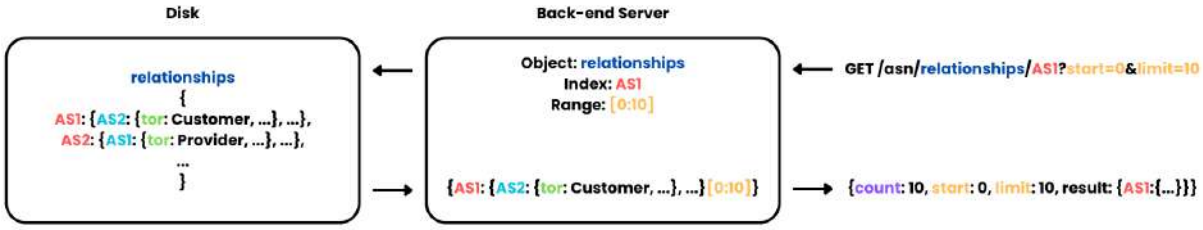

Figure 12. Back-end server workflow example

5.3. Front-end

The front-end is implemented as a web application using the framework Streamlit, in Python. Its interface is modeled after a search engine, offering a search bar through which users can input queries and receive the corresponding information, if it exists in the system's database.

As previously mentioned, Streamlit is unique in the sense that it implements an architecture a bit different from the usual front-end development. Usually, dynamic web pages or web applications are implemented from the perspective of the client, in the sense that the code written by the developer is what will execute on the client's browser. This code, the web page, needs to be served by a web server, and, once in the client's browser, it can connect to other external services, such as a back-end, and display the relevant contents to the user.

The way Streamlit generally works is that, as a developer, you implement a Python script that describes your web page using various of the framework's functions. The framework then will be responsible for actually generating the web page that will be served to the client. The point is that the code you write to describe the web page will run on the Streamlit server, and not on the client's browser. To illustrate, one consequence of this is that

the back-end does not need to be exposed, since clients will not be directly connecting to it, but the Streamlit server, which lives on the same local network, will. Following this architecture, most of the processment described next is implemented on the level of the Python script and, thus, runs on the Streamlit server.

The front-end currently accepts four types of queries: an AS number, the name of an AS set, the name of a route set and a network address prefix. Upon receiving the query, it does a simple validation based on regular expressions to check whether it is valid and, if it is, to which type it refers to. If it is not valid, the user simply receives a message that nothing was found. If it is, the server then proceeds to contact the back-end to retrieve all the relevant information. This process may involve various calls to various endpoints. For instance, the results of a search for an AS include relationships, set membership and registered addresses, each stored in its own object and accessible through its own endpoint.

Before requesting the whole extent of the information, the front-end first consults a special object of the back-end, the "metadata" object, which contains the keys of all the entities in the database. The back-end server may not contain data regarding the specific entity searched for, in which case the front-end will identify the situation through this first request and proceed to inform the user. This is a mechanism to prevent weird partial results, which would be more complex to deal with.

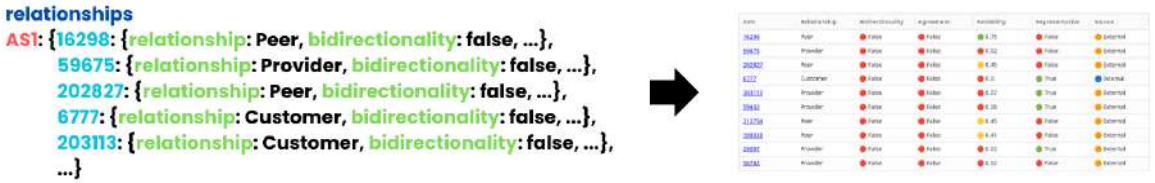

Figure 13. Example of creating a visualization from the raw data

Most of the data is displayed to the user in the form of tables. The Streamlit server, once having received a query, consulted the back-end and received back the relevant information in the form of a collection of JSON objects, will proceed to convert each one of them into a table and display it to the user. Table manipulation was made with the aid of the Pandas library, while the table visualization was made with one extension of Streamlit based on the JavaScript framework AG Grid. Some parsing is applied to the results in order to introduce some visual elements to their values, such as an icon indicating "low", "medium" and "high" thresholds for numerical values. An illustration of the conversion of a JSON object into a table is presented in Figure 13.

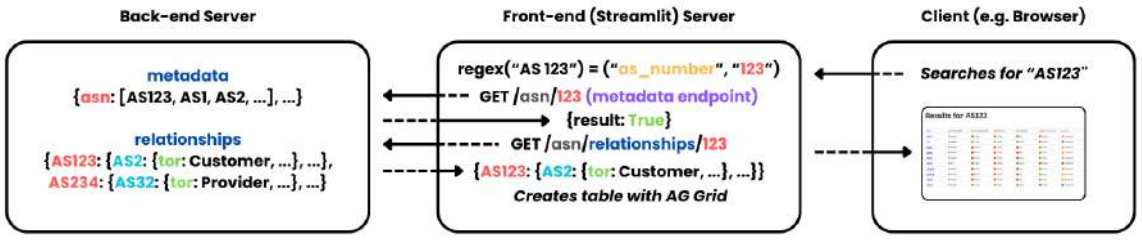

Figure 14. Front-end server workflow example

An illustrative example of a full interaction with a user is shown in Figure 14. The user consults information about AS 123, for which the front-end will display only information about its relationships. First, the front-end server receives the query "AS 123",

identifies that it corresponds to an autonomous system and that its key is “123”. Then, it sends a request to the metadata endpoint of the back-end server to check whether the database contains information about such AS. The back-end server responds that it does, and the front-end server proceeds to request for the relationship information. Upon receiving, it parses it into a table and sends the visualization to the user, concluding the initial request.

6. Deployment

rpslweb was deployed utilizing the Docker container technology within a virtual machine hosted on the Federal University of Minas Gerais servers. A schematic of the infrastructure of deployment is presented in Figure 15.

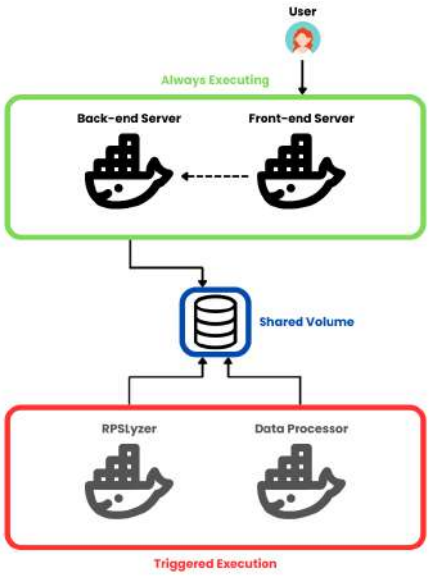

Figure 15. Infrastructure of the deployment of rpslweb

The whole system is organized as a set of four containers. Three of them refer to each of the three modules that fundamentally make up the system — the data processor, the back-end server and the front-end server —, while the fourth one executes RPSLyzer. The back-end and front-end server containers are executed continuously to make the application available. Meanwhile, the RPSLyzer and data processor containers are executed periodically in order to keep the system’s data up-to-date. This is accomplished by means of cronjobs. A job was configured to execute a script periodically, which would put both containers to execute. After they are finished, they just exit. It is important to note that RPSLyzer’s container must be executed first for the data processor container to have data to work with. Both of these containers and the back-end are connected through a shared volume, on which all the data used by the system is stored.

rpslweb is accessible through this link, and the source code of the whole project can be found on this Github repository. The system can be executed locally, either through the containers or manually deploying each component separately. Some modifications were necessary to be applied on RPSLyzer’s code, so a copy of it was included in *rpslweb*’s repository, but all install instructions can still be found on its original repository.

7. Related Work

There are various works related to automated tooling based on the IRR — and, therefore, in RPSL policies. As an example, "bgpq4" is an open-source tool that generates various configuration files, such as prefix lists and AS-paths lists. It uses data gathered from the distributed databases of the IRR and is currently being actively maintained [8].

"IRRToolSet" is another open-source tooling repository that aims at making available a set of tools with which operators can work with in regard to the data from the IRR. It assists with tasks such as configuration, verification and maintenance. Unfortunately, it is no longer actively maintained and, on that note, the Github page redirects its users to the "bgpq4's" web page [9]. The biggest problem observed with such tools is that they rely directly on RPSL policies and their databases, which, as mentioned, can be complex, vary in detail, contain inconsistencies and be susceptible to attacks.

The "IRRd", or "Internet Routing Registry Daemon", is a database server for the IRR, which covers extensive manipulation of its infrastructure, including querying, mirroring of different hosts and authorization mechanisms. It is currently maintained by Reliably Coded, a personal company from the Netherlands, in cooperation with various other organizations, such as NTT, ARIN and RIPE [10]. Again, this tool deals directly with raw RPSL policies and, therefore, is susceptible to all of the previously discussed downfalls.

The closest technology found with respect to *rpslweb* is "IRRExplorer", a website that allows queries to IRR data in a unified web interface, as well as makes available additional aggregated information, such as inconsistencies and potential problems [11]. In fact, the core of both systems is basically the same, but this project is aimed at developing a tool that empowers parsed RPSL data instead of raw RPSL data. It is believed that new insights can be built on top of the parsed information, as RPSLyzer has been proved to be more general than most tools and to be able to deal with more complex RPSL specifications [2]. Given the proven heterogeneity and inconsistency of RPSL usage, it is believed to be a more comprehensive and better quality data.

The main author of RPSLyzer, Sichang He, has made an attempt to make the data of parsed RPSL policies and BGP route verifications available as a PostgreSQL database queryable through a REST interface [12]. According to him, the project was just part of a database course, and also, for the main proceedings of the RPSLyzer's article, operations had to be done in memory, and not in a database, otherwise they would be too slow. After both course and paper production, which were the main focus, had finished, it was abandoned.

In terms of RPSL data analysis and insight inference, there is a plethora of works that tackle such tasks. As a first example, in the article "IRRegularities in the Internet Routing Registry", Du et al. propose a framework to identify irregular route objects published in the IRR through the comparison of data from different database providers, as well as from BGP announcements and RPKI records [7]. Unfortunately, the data produced by RPSLyzer is constructed over single copies of each object, so that a comparison across different database providers, or in relation to BGP and RPKI, would require the acquisition of additional information from external sources, which is outside the scope defined for this project.

Another interesting work, already previously presented and discussed, but included here as a reference, was proposed by researchers of the University of Tel-Aviv, in the article "IRR-Based AS Type of Relationship Inference". In it, the type of relationship between ASes, such as Provider-to-Customer (P2C), Peer-to-Peer (P2P) or Siblings (S2S), is inferred based on RPSL objects stored within the IRR. The methodology is constructed over a series of

heuristics that leverage the information contained within these objects, mainly “aut-num” and “as-set” objects, in order to infer the type of relationship between pairs of ASes. A metric is also proposed to evaluate the reliability of each inference across heuristics, as a means to parametrically filter them according to stronger evidence of correctness. The results obtained are compared against manually tagged datasets, revealing a reasonable degree of accuracy [13]. This work, as mentioned, is especially interesting since it is based solely on single objects out of the IRR, so that the data from RPSLyzer can be used to reproduce its results.

8. Conclusion

This document was aimed at being a technical specification of $rpslweb$, a web platform that makes available useful information regarding data contained within the IRR, empowered by the parsing capabilities of RPSLyzer. Beyond main features, architectural decisions and other technical details, it also presents some theoretical background for relevant concepts. In the end, a working release of the web application was successfully developed and deployed, as was envisioned by the goals defined for this project, which represents a step forward towards the ultimate goal of improving the integrity of the Internet as a whole.

9. Future Work

In this section, we present a discussion about possible points of improvement in regard to $rpslweb$. Even though the objective established for the project was accomplished, that is, the implementation of a web platform that makes available useful internet routing information, there is still substantial margin for improvement and extensions.

First, as can be seen in the section [Layout and Features], while the results for a search for an autonomous system are rich, searches for other objects, such as AS sets or address prefixes, still return limited information. Also, not only that, but some data produced by RPSLyzer, such as peering and filter sets, is currently ignored. Therefore, in future iterations of this work, there is still margin to expand the amount of information that can be acquired through the platform in regard to many dimensions of the data contained in the IRR.

Second, as was mentioned in the subsection [Data Processor], during the cleaning subtask, all information that was deemed inconsistent was simply thrown away. Identifying inconsistencies in the RPSL policies stored within the IRR is a relevant task by itself, so that such information, as well as the output of RPSLyzer — which also identifies inconsistencies in the parsed specifications, but was not used —, would also be an interesting and useful detail to display within the platform.

Finally, it is noted that the system’s infrastructure leans towards simple, yet less robust technologies. This simplicity, in the current scope, was beneficial, since it made tasks faster and easier to complete. However, in the future, if the system were to become popular, the lack of robustness would become a bottleneck. Therefore, in such a scenario, it would be interesting to utilize more comprehensive technologies, such as proper object storage (e.g. Redis) for the data or an orchestration system (e.g. Kubernetes) for container management.

This discussion by no means is trying to be exhaustive, as there is probably a substantial amount of improvement points for the platform, but it aims at providing an overview, or a first guidance, that should be useful for future iterations of this project.

Artificial intelligence (AI) tools were **not used for any purpose during the elaboration of this project.*

Bibliographic References

[1] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, M. Terpstra, “Routing Policy Specification Language (RPSL)”, RFC *2622* [Online], June 1999. Available: https://www.rfc-editor.org/rfc/rfc2622

[2] S. He, I. Cunha, E. Katz-Bassett, "RPSLyzer: Characterization and Verification of Policies in Internet Routing Registries," in *Proceedings of the 2024 ACM on Internet Measurement Conference*, 2024, pp. 365–374.

[3] L. Peterson, B. Davie. (2019, Nov 26). *Computer Networks: A Systems Approach* (sixth edition) [Online]. Available: https://book.systemsapproach.org/

[4] Merit. (2025, Apr 19). *Internet Routing Registry* [Online]. Available: https://irr.net/

[5] Datareportal. (2025, Apr 19). *Digital Around the World* [Online]. Available: https://datareportal.com/global-digital-overview

[6] S. He, I. Cunha. (2024, Nov 26). *RPSLyzer: Parse RPSL Policies and Verify BGP Routes* [Online]. Available: https://github.com/SichangHe/internet_route_verification

[7] Du, B., et al, "IRRegularities in the Internet Routing Registry," in *Proceedings of the 2023 ACM on Internet Measurement Conference*, 2023, pp. 104–110.

[8] A. Snarskii, C. David, C. Jeker, J. Snijders, M. Stucchi, M. Litvak, P. Schoenmaker, R. Wichertjes. (2025, Apr 6). *Bgpq4* [Online]. Available: https://github.com/bgp/bgpq4

[9] N. Hilliard, F. Liambotis, A. Sheshka, J. Snijders, R. Thorpe. (2023, Dec 4). *IRRToolSet* [Online]. Available: https://github.com/irrtoolset/irrtoolset

[10] Reliably Coded. (2025, Mar 21). *Internet Routing Registry Daemon (IRRd) Version 4* [Online]. Available: https://github.com/irrdnet/irrd

[11] S. Romijn, T. Vink, J. Snijders, C. de Reuver, T. de Kock. (2025, Apr 3). *IRR explorer: explore IRR & BGP data in near real time* [Online]. Available: https://github.com/nlnog/irrexplorer

[12] S. He. (2023, Dec 6). *Internet Route Verification Server* [Online]. Available: https://github.com/SichangHe/internet_route_verification_server

[13] A. Zulan, O. Miron, T. Shapira, and Y. Shavitt, *IRR-Based AS Type of Relationship Inference*. 2025. [Online]. Available: https://arxiv.org/abs/2504.10299

[14] L. Blunk, J. Damas, F. Parent, A. Robachevsky, “Routing Policy Specification Language next generation (RPSLng)”, RFC *4012* [Online], March 2005. Available: https://www.rfc-editor.org/rfc/rfc4012